

Software Architecture Design Domain

S. A. White

Repository Based Software Engineering
University of Houston - Clear Lake
Houston, TX 77058

ABSTRACT

Software architectures can provide a basis for the capture and subsequent reuse of design knowledge. The goal of software architecture is to allow the design of a system to take place at a higher level of abstraction; a level concerned with components, connections, constraints, rationale. This architectural view of software adds a new layer of abstraction to the traditional design phase of software development. It has resulted in a flurry of activity towards techniques, tools, and architectural design languages developed specifically to assist with this activity. An analysis of architectural descriptions, even though they differ in notation, shows a common set of key constructs that are present across widely varying domains. These common aspects form a core set of constructs that should belong to any ADL in order to for the language to offer the ability to specify software systems at the architectural level. This analysis also revealed a second set of constructs which served to expand the first set thereby improving the syntax and semantics. These constructs are classified according to whether they provide representation and analysis support for architectures belonging to many varying application domains (domain-independent construct class) or to a particular application domain (domain-dependent constructs). This paper presents the constructs of these two classes, their placement in the architecture design domain and shows how they may be used to classify, select, and analyze proclaimed architectural design languages (ADLs).

1. INTRODUCTION

Many software architecture design languages have been defined, some for use within specific application domains (for example, Meta-H , DICAM and Control-H (Kogut et al, 1995), Rapide (Luckham et al., 1995), and others more general (for example UniCon (Shaw et al., 1995), KAPTUR (Bailin, 1992). If a language is to be classified as an architectural design language there should be something concrete that can be measured by

inspection of the language constructs that tell us whether the language indeed belongs to this category. Of course, the syntax is irrelevant, but the construct type and its ability to provide the language with the required architectural level specification is what is important. For example, all programming languages must provide some type of sequence, selection, and repetition control constructs (Pratt and Zelkowitz, 1996). The syntax, ease of use, ease of implementation, and other considerations affecting instances of these construct types provide further means of classification and selection of an appropriate language for the programming task at hand. The sequence, selection, and repetition construct types are part of the domain of programming language design. In particular they belong to the domain-independent set of construct types since they belong to all programming languages in some form or another. The parallel is also true for architecture languages. Architectural languages are also defined by an architectural design domain that provides a set of domain-independent construct types that must be present in all architectural languages. This paper reports on the definition of the domain-independent construct types (referred to as the "core set" in this paper) and the relationship of this set to the set of domain-dependent constructs (referred to as the "perimeter set") within the architectural design language domain.

In the sections to follow these aspects of the software architecture design domain are presented: 1) definition of software architecture, 2) definition of architecture description languages including how it differs from standard design notions and 3) the identification and definition of two general classes of constructs (the core and perimeter classes) that have been found to belong to the domains of both logical and physical product design, and the use of this classification as a architecture language design and evaluation technique.

2. SOFTWARE ARCHITECTURE

Software Architecture has been defined in varying ways. Perry and Wolf (1992) describe a software system's architecture as a set of elements, form, and rationale. This definition is one of the earliest and was derived after an examination of architectures from other domains such as building construction, computing hardware, and network architectures. Perry and Wolf also define an *architectural style* as "that which abstracts elements and formal aspects from various specific architectures. An architectural style is useful for addressing classes of design decisions rather than specific instances of design decisions that relate to only a particular architecture (Garlan and Shaw, 1993). Both a style and a specific architecture instance can and should be represented by at least a processing view, a data view, and a connections view (Perry and Wolf, 1992).

Garlan and Shaw (1993) consider software architecture to consist of components, connectors, and configurations. In this definition components are computational elements, connectors are descriptions of the interactions between these components, and configurations are the resulting structure of topology of the architecture which can reflect the attributes of a particular architectural style. Garlan denotes architectural style as defining a family (class) of architectures that share a common vocabulary of components and connectors, and which meet a set of constraints on the topology of that style (Garlan et al., 1994). Rationale is not a part of Garlan and Shaws definition although it is considered an essential aspect by others (Tracz 1995; Perry and Wolf, 1992).

Many other definitions abound, but the above seem to be influencing most definitions. We adopt the following definition: a software architecture consists of a configuration of software components, connectors, connections and a set of associated constraints and rationale collected into one universal view or preferably spread across multiple views of the architecture. Each view will serve to capture a level of abstraction that reflects a view of components, connections, constraints, and rationale tailored for a particular user. Users include those involved in the software product development cycle such as the customer, the user/stakeholder, the software architect, the software developer, and maintainer. Indeed, even for each user/stakeholder there may exist multiple views (Gacek 1995). For example, for the software architect stakeholder the data view, processing view, connections view, and the universal or combined view are all important abstractions of the software design that provide different types of information that need to

be represented and analyzed before a detailed design is to begin.

3. ARCHITECTURE DESCRIPTION LANGUAGES (ADLS)

According to Clements and Kogut (1995), an ADL is a set of notations, languages, standards and conventions for an architectural model. More simply stated an Architectural Description Language (ADL) is a language that is designed specifically for the representation and analysis of software architectures. It defines a formal syntax for expressing architectures and assigns a semantic interpretation. Some examples of ADLs include MetaH, DICAM and Wright (Ahmed 1995), KAPTUR (Bailin, 1992), Rapide (Luckham et al., 1995), and UniCon (Shaw et al, 1995). Of course, each language differs in its intended use and audience and therefore its syntax and semantics differ greatly. Some are very specialized so as to be useful only in a particular application domain, such as Rapide and MetaH. Others tend toward more general use such as UniCon and Wright, while others provide a method for generating a domain-specific design language (DFR (White, 1996), SDFR (White, 1995)) from a description of the design domain.

3.1 Differences between ADLs and standard design notations

ADLs differ from standard system design notations in that the former tend to support components of larger granularity and are more often tailorable to given domains (Clements and Kogut, 1995). They, like design notations, provide models of the solution space. However they differ from design notations in that they (ideally) provide constructs to support reuse /megaprogramming, rapid prototyping, connections as first class entities, and rationale for constraints imposed on architectural components and connections (Luckham and Vera, 1995; Dean and Cordy, 1995). They provide for a larger and more encompassing view of the software than standard design notations while at the same time providing, through various architectural views, enough detail for analysis and evaluation. Depending on the particular architectural view (data, structural, behavioral, or stakeholder oriented view) there can occur varying levels of overlap with what may appear in a traditional design notation developed to deal with that particular view. However, the type of components and connections that are being represented will in general greatly differ in abstraction level from the design elements considered in the traditional design notations. Furthermore, the actual placement of the software architecting process within the software

lifecycle further distinguishes it from the classical design phase. In essence, the software architecting process has expanded the traditional design phase to include a phase that immediately precedes the traditional high-level design phase and will at times overlap this phase depending on the architectural view.

4. ADLS: THE CORE SET AND PERIMETER SETS.

In this section we identify and define the general construct types that belong to the architecture design domain and present a method of classification of these types. This classification forms a model of the architecture design domain that can be used when defining or evaluating an architecture design language.

We separate the class of constructs that a ADL should possess into an *core* (fixed) set and a *perimeter* (variable) set. The core set is defined by a class of constructs that we have found to be present across widely varying design domains (White 1995a; White 1995). These construct types were found to exist in physical product domain description languages (e.g., FDDL (Gu, 1989) and DFR (White 1995a) and SDFR (White 1995) as well as within existing logical domain description languages (UniCon, Rapide, Aesop (Garlan et al, 1994)). These languages represented widely varying design environments yet when they were viewed from the perspective of each sharing the common goal of specification of the assembly (product structure) and manufacture (implementation) of a artifact, be it physical or logical, the commonalities of the essence of this type of specification surfaced¹. These commonalities are captured by the construct-types found to belong to the core set and the core features of these types. This set defines an abstract class of constructs that capture the nature of the abstraction that architecture languages should provide. The perimeter set defines the way in which the core set is extended by defining subtypes of the core types and by defining new features of the core types.

The core set represents the domain-independent features of the architectural design domain. As such, the core set defines those constructs that must belong to any ADL in order for that language to be classified as a

¹ Logical design is design activity that deals with the creation of an artifact that is itself an abstraction of the real world, i.e., software. Physical design is design activity that deals with the creation of actual physical objects, i.e., apparel, furniture, appliances, etc.

valid architectural design language. The perimeter set, on the other hand is that part of the architectural design domain that will vary depending on the design domain (product-line) the ADL is to be used for. In each different domain the perimeter set will extend the core set to allow a more domain-specific vocabulary as well as provide for a set of syntactic and semantic analysis rules that can provide a deeper level of analysis due to the assumptions (built-in semantics) the language can make since it is designed for use in a particular design situation. The more specific the language, the larger and more precise the semantics associated with the perimeter set will be. Both the core set and perimeter set are defined by construct types which have syntax and semantics associated with each type.

4.1 The core set.

The core set C is defined by a set of abstract construct types {Components, Connections, Configuration, Rationale, Constraints}. These abstractions exist in most definitions of software architecture and indeed they were found to be present in some form across both physical and logical design domains - with the exception of rationale which was missing from many. These types are further defined by a set of attribute types that define the core features of the core constructs. These attribute types are listed in Table 1 along with the construct types they are defined on and an explanation of what type of specification this combination allows (what the general semantics are).

The implication of the existence of the core set is that we can assert that an ADL should possess an instance of each of the construct-types defined in the core set. In other words, there should be construct types in the ADL that function semantically as subtypes of the core set types. As such, this core set provides a method for the evaluation of existing ADLs and a starting point for the design of new ADLs. We use this core set to define a set of evaluation criteria that can be applied to an ADL definition to determine if the language will support the basic tenets of software architecture design. That is, it should be possible to express architectural designs that result in the improvement of software reuse, reengineering, rapid prototyping and maintenance activities. This evaluation criteria is presented in Table 2. The left-hand column of the table lists the construct-type the requirement applies to and the right-hand column lists the question to ask of the language. Note that the requirement is not explicitly stated but is implied by the question.

Table 1 The core set

Core Construct	Core Attribute	Specification and explanation
Component / Connection	Offshelf/implement	Specifies if construct is to be taken from a library of components or if it is to be manufactured.
Component / Connection	Required / optional	Specifies if the construct is required or optional.
Component / Connection	Ports	Describes the architectural interface.
Component / Connection	Role	Classifies the functionality.
Component / Connection	Sub-components	Lists all immediate sub-components that are needed to construct a compound component.
Component	Simple, Compound, Offshelf.	Classifies the component as belonging to one of these three general classes.
Component / Connection	Type-classification	Specifies the type of component. Could belong to one of a set of built-in types, or user-defined types.
Component / Connection	Control-view	Specify control flow information.
Component / Connection	Data-view	Specify data flow information.
Component / Connection	Rationale	Specify rationale for the particular component / connection.
Component / Connection	Operations	Specify high level operations.
Configuration	Connector	Specifies the topology.
Rationale	who, what where, when , why, why	Provides for a textual explanation. Can be associated with all construct types.
Component	Connect -to	Specifies the architectural configuration.
Connection	Connects-components	Specifies the architectural configuration

4.2. The perimeter set

For any domain-specific ADL, there will exist a set of language constructs that lie outside the set of core constructs, C . We call this set the perimeter set, p . The perimeter set is defined by a set of attribute types and construct types that serve to extend the core set in order to provide support for domain specific representation and analysis that is not fully addressed by the general attributes of the core set. Each perimeter set is domain dependent and reflects the architectural style(s) that is (are) used in that domain. For example, if the application domain is found to contain a pipe-filter style (Shaw et al., 1995) then the component-type “pipe” and the connection-type “filter” may appear in the language definition as primitive construct types in the language. However, these two construct types are merely subtypes of component and connection types. There may also be attributes associated with the new pipe and filter types

that allow them to be specified beyond that provided by the attributes of the component super-type. Once the perimeter set is defined for a given domain it can be used in conjunction with the core set evaluation to determine if the given ADL contains the constructs necessary to support a deeper level of architectural design and analysis activity than that provided by the core set alone.

Therefore for a domain-specific ADL the total set of constructs, T , can be defined as $T = C \cup p_i$ where p_i denotes the particular perimeter set i of construct types that are necessary to represent and analyze architectures within domain i . If the ADL is designed to cover n domains, then the n -domain perimeter set, P , is the union of all p_i where $i \in \{1, n\}$. The set of construct-types T in a n -domain ADL is defined as: $T = C \cup P$.

4.3 Use of these two sets.

The core and perimeter set have provided the basis for the design and implementation of a prototype that allows a design language domain to be specified by providing for the specification of subtypes of the core set and by allowing the specification to vary according to the specification rules defined in the n-domain perimeter set, P. The prototype domain model is based on the rules in C and P. Use of the tool provides for the specification of an instance of a particular perimeter set, p, which results in a domain specific design language definition such that $T = C \cup p_p$. This tool is actually a domain specification language that allows a design language domain to be defined. The semantics of this language perform the generation of the actual domain specific language and its syntactic and semantic analyzer. Thus, automatic generation of a specific design language is one way in which we have used the existence of these two sets. In short, these two sets provide the definition of the types of information that should be included in a definition of a specific architectural design domain.

5. SUMMARY AND CONCLUSIONS

An analysis of architectural descriptions, even though they differ in notation, shows a common set of key constructs that are present across widely varying domains. These common aspects form a core set of constructs that should belong to any ADL in order to for the language to offer the ability to specify software systems at the architectural level. This analysis also revealed a second set of constructs which served to

expand the first set thereby improving the syntax and semantics so domain specific design rules could be built into the semantics of the language. All software architecture design languages belong to the software Architecture Design Language domain and therefore should contain the equivalent of the construct types defined by the core set of this domain to qualify as an architecture language. The core set may be augmented in any particular instance of a software architecture domain (a particular software architecture language) by perimeter sets that are determined by the application domain at hand. That is, there should be a fixed class of constructs that belong to all architectural definition domains, but there will also exist a secondary class of constructs that may or may not be present depending on the application domain at hand. Identification and study of these two classes of constructs will allow for the construct classification of the software architectural design domain to be defined. This definition can now be utilized in the creation of architectural design languages. It provides the beginning of a domain model for architectural language design and insight into separation of a design language from an ADL and the construct types necessary for such distinction.

ACKNOWLEDGMENTS

This work was partially supported by the RBSE project under the auspices of RICIS at the University of Houston - Clear Lake, by the Institute for Space Systems Operations (ISSO) University of Houston, and the Faculty Research Support Fund of UHCL.

Table 2 Core-set based evaluation

Construct	Criterion
Components -	Does the language provide a set of built-in component types? If yes, do these built-component types provide the required level of abstraction to be considered useful at the architectural level of design? Does the language provide a set of built-in component attributes that allow for specification of the necessary architectural components?
Components -	Does the language allow for new components to be defined as instances of built-in component types? If yes, does the language allow for additional constraints to be specified on the new component as long as the ancestor constraints are not violated? Does the language provide the ability to compose built-in component types to form new connection types?
Connections -	Does the language provide a set of built-in connection types? If yes, do these built-connection types provide the required level of abstraction to be considered useful at the architectural level of design? Does the language provide a set of built-in connection attributes that allow for specification of

	the necessary architectural connections?
Connections -	Does the language allow for new connections to be defined as instances of built-in connection types? If yes, does the language allow for additional constraints to be specified as long as the ancestor constraints are not violated? Does the language provide the ability to compose built-in connection types to form new connection types?
Constraints -	Does the language provide built-in constraints upon the type of components and connections that may be used in a particular architectural style? Does the language provide built-in constraints upon the type of configurations that may be used between specific component types?
Constraints -	Does the language provide special syntactic constructs that allow a constraint to be specified with regard to the reusability of any component or connection defined by the software architecture? If yes, does the language provide semantic checks that are able to enforce these constraints on the architecture design?
Rationale - why, when, where, who, what .	Can rationale for the need and use of each component, connection, and constraint be made? Can it be made formally , informally or both ?
Components, Connections, and Configurations	Is syntax analysis provided? Is semantic checking of built-in restrictions on component and connection usage provided? Is semantic checking of user-defined constraints provided? Can completeness and consistency be checked?
Configurations	Can two architectural descriptions be compared and the similarities and differences extracted? Can separate views of the architecture be analyzed individually?
Components, Connections, and Configurations	Can the language allow separate views to be described, such as a data view, structural view, control view, process view or other? Can two architectural descriptions be compared and the similarities and differences extracted?

REFERENCES

- Ahmed A. Abd-Allah, 1994 "Architecture Description Languages," Slide Presentation, University of Southern California, Center for Software Engineering, June 1994, slides 1-36.
- Victor R. Basili, L.C. Briand, W.M. Thomas, 1994 "Domain Analysis for the Reuse of Software Development Experiences", *Proceedings of the 19th Annual Software Engineering Workshop*, NASA/GSFC, Greenbelt, Maryland, December, pages 1-14.
- Victor R. Basili, Gianluigi Caldiera and H. Dieter Rombach, 1994, "The Experience Factory", Encyclopedia of Software Engineering, Wiley & Sons, Inc.
- Sidney C. Bailin, "KAPTUR: A Tool for the Preservation and Use of Engineering Legacy", CTA Incorporated, 1992.
- Lou Coglianese, Mark Goodwin, Roy Smith, Will Tracz, Don Batory, Kirstie Bellman, David Gries, David McAllestar, Rick Selby, and Richard Taylor, 1992, "An Avionics Domain-Specific Software Architecture", Special Report CMU/SEI-92-SR-9, Carnegie Mellon University, Software Engineering Institute.
- Thomas R. Dean and James R. Cordy, 1995, "A Syntactic Theory of Software Architecture", *IEEE Transactions on Software Engineering*, 21(4), April, pages 302-313.
- David Garlan and Mary Shaw, 1993 "An Introduction to Software Architecture", Technical Paper CMU/SEI-93-TR-33, Carnegie Mellon University, Software Engineering Institute.
- Cristina Gacek, 1995 "Exploiting Domain Architectures in Software Reuse", *Proceedings of the ACM-SIGSOFT Symposium on Software Reusability (SSR'95)*, ACM Press, Seattle, Washington, 28-30 April, pages 229-232.
- Cristina Gacek, Ahmed Abd-Allah, Bradford Clark and Barry Boehm, 1994, "Focused Workshop on Software Architectures: Issue Paper" *Knowledge Summary of the USC-CSE Focused Workshop on Software Architectures*, University of Southern California, Center for Software Engineering, June.
- P. H. Gu, 1989, "Artificial Intelligence Approach to Integration of Feature Based Design and Manufacturing Tasks Planning", PhD Thesis, McMaster University, 1989.
- David Garlan, Robert Allen, and John Ockerbloom, 1994, "Exploiting Style in Architectural Design Environments", *Proceedings of the ACM SIGSOFT '94 Symposium on Foundations of Software Engineering*, New Orleans, Louisiana, December 1994, pages 1-13.
- Frederick Hayes-Roth, Lee D. Ertman and Allan Terry, 1992, "Domain-Specific Software Architectures: Distributed Intelligent Control and Communication", Special Report CMU/SEI-92-SR-9, Software Engineering Institute, Carnegie Mellon University, pages 29-63.
- Paul Kogut and Paul Clements, 1995, "Features of Architecture Representation Languages", *Proceedings of the Seventh Annual Software Technology Conference*, Salt Lake City, Utah, Air Force Software Technology Support Center, April, pages 1-39.
- David Luckham and James Vera, 1995 "An Event-Based Architecture Definition Language", DARPA technical report under ONR contract N00014-92-J-1928 under Grant AFOSR92-0354, pages 1-20.
- David Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan and Walter Mann, 1995, "Specification and Analysis of System Architecture Using Rapide", *IEEE Transactions on Software Engineering*, 21(4), April 1995, pages 336-353.
- Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young and Gregory Zelesnik, 1995, "Abstractions for Software Architecture and Tools to Support Them", *IEEE Transactions on Software Engineering*, 21(4), April, pages 314-334.
- Terrance W. Pratt and Marvin V. Zelkowitz, 1996, "Programming Languages Design and Implementation", 3rd edition, Prentice Hal, Englewood, NJ.
- D. E. Perry and A. L. Wolf, "Foundations for the Study of Software Architecture", *ACM SIGSOFT'92, Software Engineering Notes* Vol. 17. No. 4. pp. 40-45.
- Will Tracz, 1995 "DSSA (Domain-Specific Software Architecture) Pedagogical Example", *ARPA Domain-Specific Software Architecture (DSSA) Program*, Loral Federal Systems, URL: <http://www.sei.cmu.edu/arpa/dssa/DSSAexp.html>.
- S. A. White, 1996a "A Design Metalanguage for Design Language Creation", *Proceedings of ASME and API Energy Information Management - Incorporating ETCE, (ASME-ETCE96)* Houston TX, Jan. 29 - Feb

2, 1996, Volume I Computers in Engineering, pp. 135-144.

S. A. White, 1995, "A Framework for the development of Domain Specific Design Support Systems", *Proceedings of the First World Conference on Integrated Design & Process Technology*, Austin, TX., IDPT- Vol 1. pp. 37-42, Dec. 6-9, 1995.

S. A. White, 1995a "Development of a Language-based Framework for the Automatic Generation of Domain-specific Design Languages", *Proceeding of the ASME Symposium on Computers in Engineering, (ASME-ETCE95)* , Houston TX, Jan. 29 - Feb 1., PD vol. 67. pp. 23-32.

S. A. White and L. E. , 1994, "Albright Integration of the Design and Realization Phases of Apparel Manufacturing", *Proceedings of the Fifth Annual Academic Apparel Research Conference (AARC)*, Lafayette LA., Feb. 17-18, pp. 5-1:5-13.

S. A. White, 1994, *A framework for the Integration of the Design and Realization phases of Product Development*, S. A. White, Ph.D. thesis. Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, LA. May.

